Parallel Volume Rendering Using Binary-Swap Image Composition

Kwan-Liu Ma

ICASE, NASA Langley Research Center

James S. Painter

Department of Computer Science, University of Utah

Charles D. Hansen

Michael F. Krogh

Advanced Computing Laboratory, Los Alamos National Laboratory

Abstract

This paper presents a divide-and-conquer ray-traced volume rendering algorithm and a parallel image compositing method, along with their implementation and performance on the Connection Machine CM-5, and networked workstations. This algorithm distributes both the data and the computations to individual processing units to achieve fast, high-quality rendering of high-resolution data. The volume data, once distributed, is left in place, regardless of viewing position. The processing nodes perform local raytracing of their subvolume concurrently. No communication between processing units is needed during this local ray-tracing process. A subimage is generated by each processing unit and the final image is obtained by compositing subimages in the proper order, which can be determined *a priori*. Composition is done in parallel via a new algorithm we call *Binary-Swap compositing*. Test results on both the CM-5 and a group of networked workstations demonstrate the practicality of our rendering algorithm and compositing method.

1 Introduction

Existing volume rendering methods, though capable of making very effective visualizations, are very computationally intensive and therefore fail to achieve interactive rendering rates for large data sets. Although computing technology continues to advance, computer processing power never seems to catch up with the increases in data size. Our work was motivated by the following observations. First, volume data sets can be quite large, often too large for a single processor machine to hold in memory at once. Moreover, high quality volume renderings normally take minutes to hours on a single processor machine and the rendering time usually grows linearly with the data size. To achieve interactive rendering rates, users often must reduce the original data, which produces inferior visualization results. Second, many acceleration techniques and data exploration techniques for volume rendering trade memory for time, which results in another order of magnitude increase in memory use. Third, motion is one the most effective visualization cues, but an animation sequence of volume visualization normally takes hours to days to generate. Finally, we notice the availability of massively parallel computers and the hundreds of high performance workstations in our computing environments. These workstations are frequently sitting idle for many hours a day. All the above lead us to investigate ways of distributing the increasing amount of data as well as the time-consuming rendering process to the tremendous distributed computing resources available to us.

In this paper we describe the resulting parallel volume rendering algorithm, which consists of two parts: parallel ray-tracing and parallel compositing. In our current implementation on the CM-5 and networked workstations, the parallel volume renderer evenly distributes data to the computing resources available. Without the need to communicate with other processing units, each subvolume is ray-traced locally and generates a partial image. The parallel compositing process then merges all resulting partial images in depth order to produce the complete image. Our compositing algorithm is particularly effective for massively parallel processing as it always makes use of all processing units by repeatedly subdividing the partial images and distributing them to the appropriate processing units. Our test results on both the CM-5 and workstations are promising, and expose different performance issues for each platform.

2 Background

An increasing number of parallel algorithms for volume rendering have been developed recently [1, 2, 3, 4]. The major algorithmic strategy for parallelizing volume rendering is the divide-and-conquer paradigm. The volume rendering problem can be subdivided either in data space or in image space. Data-space subdivision assigns the computation associated with particular subvolumes to processors, while image-space subdivision distributes the computation associated with particular portions of the image space. Data-space subdivision is usually applied to a distributed-memory parallel computing environment while image-space subdivision is often used in shared-memory multiprocessing environments. Our method, as well as the similar methods developed independently by Hsu, Camahort and Neumann [1, 2, 3], can be considered hybrid methods because they subdivide both data space (during rendering) and image space (during compositing).

The basic idea behind our algorithm and the other similar methods is very simple: divide the data up into smaller subvolumes and distrbute the subvolumes to multiple processors, render them separately and *locally*, and combine the resulting images in an incremental fashion. The memory demands on each processor are modest since each processor need only hold a subset of the total data set. In earlier work we used this approach to render high resolution data sets in a computing environment, for example, with many midrange workstations (*e.g.* equipped with 16MB memory) on a local area network [5]. Many computing environments have an abundance of such workstations which could be harnessed for volume rendering provided that the memory usage on each machine is reasonable.

3 A Divide and Conquer Algorithm

The starting point of our algorithm is the volume ray-tracing technique presented by Levoy [6]. An image is constructed in *image order* by casting rays from the eye through the image plane and into the volume of data. One ray per pixel is generally sufficient, provided that the image sample density is higher than the volume data sample density. Using a discrete rendering model, the data volume is sampled at evenly spaced points along the ray, usually at a rate of one to two samples per voxel. The volume data is interpolated to these sample points, usually using a trilinear interpolant. Color and opacity are determined by applying a transfer function to the interpolated data values. This can be accomplished through a table lookup. Intensity is assigned by applying a shading function such as the Phong lighting model. The normalized gradient of the data volume can be used as the surface normal for shading calculations.

Sampling continues until the data volume is exhausted or until the accumulated opacity reaches a threshold cut-off value. The final image value corresponding to each ray is formed by compositing, front-to-back, the colors and opacities of the sample points along the ray. The color/opacity compositing is based on Porter and Duff's **over** operator [7]. It is easy to verify that the **over** is *associative*; that is,

$$a \textbf{ over } (b \textbf{ over } c) = (a \textbf{ over } b) \textbf{ over } c.$$

The associativity of the **over** operator allows us to break a ray up into segments, process the sampling and compositing of each segment independently, and combine the results from each segment via a final compositing step. This is the basis for our parallel volume rendering algorithm as well as recent methods by other authors [1, 2, 3, 5].

3.1 Data Subdivision/Load Balancing

The divide-and-conquer algorithm requires that we partition the input data into subvolumes. There are many ways to partition the data; Neumann compares *block*, *slab* and *shaft* data disribution [3]. Ideally we would like each subvolume to require about the same amount of computation. We would also like to minimize the amount of data which must be communicated between processors during compositing.

The simplest method is probably to partition the volume along planes parallel to the coordinate planes of the data. If the viewpoint is fixed and known when partitioning the data, the coordinate plane most nearly orthogonal to the view direction can be determined and the data can be subdivided into "slices" orthogonal to this plane. When orthographic projection is used, this will tend to produce subimages with little overlap, and therefore little communications during compositing. If the view point is not known *a priori*, or if perspective projection is used, it is better to partition the volume equally along *all* coordinate planes. This is know as *block* data distribution [3] and can be accomplished by gridding the data equally along each dimension [1, 2].

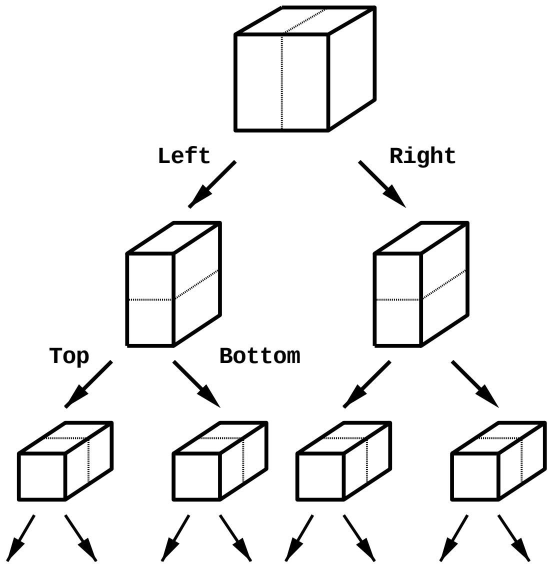

Figure 1: k-Dtree Subdivision of a Data Volume

We instead use a k-D tree structure for data subdivision [8], with alternating binary subdivision of the coordinate planes at each level in the tree as indicated in Figure 1. When the number of processors is a power of 8, the volume is divided equally among all three dimensions, and hence this is equivalent to the griding method. If the number of processors is not a power of 8, the volume will be split unevenly in the three dimensions, however, never by more than a factor of two. As shown later, the k-D tree structure provides a convenient hierarchical structure for image compositing. Note that when trilinear interpolation is used, the data lying on the boundary between two subvolumes must be replicated and stored with both subvolumes.

3.2 Parallel Rendering

Local rendering is performed on each processor independently; that is, data communications is not required during subvolume rendering. We use ray-casting based volume rendering. All subvolumes are rendered using an identical view position and only rays within the image region covering the corresponding subvolume are cast and sampled.

In principle, any volume rendering algorithm could be used for local rendering, however, some care needs to be taken to avoid visible artifacts where subvolumes meet. For example, in ray casting, we sample along each ray at a fixed predetermined interval. Consistent sampling locations must be ensured for all subvolumes so we can reconstruct the original volume. As shown in Figure 2, for example, the location of the first sample $S_2(1)$ on the ray shown in Subvolume 2 should be calculated correctly so that the distance between $S_2(1)$ and $S_1(n)$ is equivalent to the predetermined interval. Without careful attention to the sample spacing, even across subvolume boundaries, the subvolume boundaries can become visible as an artifact in the final image.

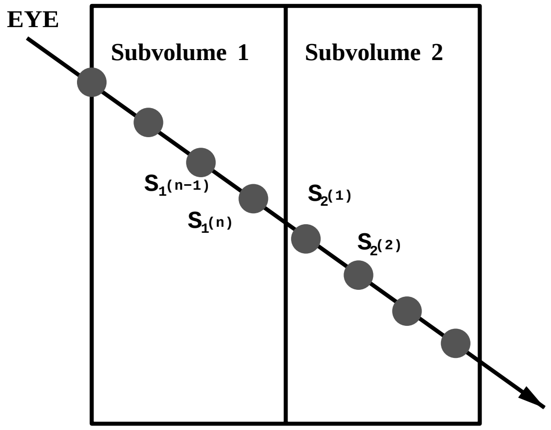

Figure 2: Correct Ray Sampling

3.3 Image Composition

The final step of our algorithm is to merge ray segments and thus all partial images into the final total image. In order to merge, we need to store not only the color at each pixel but also the accumulated opacity. As described earlier, the rule for merging subimages is based on the **over** compositing operator. When all subimages are ready, they are composited in a front-to-back order. For a straightforward one-dimensional data partition, this order is also straightforward. When using the k-D tree structure, this front-to-back image compositing order can be determined hierarchically by a recursive traversal of the k-D tree structure, visiting the "front" child before the "back" child. This is similar to well known front-to-back traversals of BSP-trees [9]. In addition, the hierarchical structure provides a natural way to accomplish the compositing in parallel: sibling nodes in the tree may be processed concurrently.

A naive approach for parallel merging of the partial images is to do binary compositing. By pairing up processors in order of compositing, each disjoint pair produces a new subimage. Thus after the first stage, we are left with the task of compositing only $n/2$ subimages. Then we use half the number of the original processors, and pair them up for the next level of compositing. Continuing similarly, after $\log n$ stages, the final image is obtained. One problem with the above method is that during the compositing process many processors become idle. At the top of the tree, only one processor is active, doing the final composite for the entire image. We found that compositing two 512x512 images required 1.44 seconds on one CM-5 scalar processor. One of our goals was interactive volume rendering, requiring subsecond rendering times, so this was unacceptable.

More parallelism must be exploited during the compositing phase. Towards this end, we have generalized the binary compositing method so that every processor participates in all the stages of the compositing process. We call the new scheme *binary-swap* compositing. The key idea is that, at each compositing stage, the two processors involved in a composite operation split the image plane into two pieces and each processor takes responsibility for one of the two pieces.

In the early phases of the binary-swap algorithm, each processor is responsible for a large portion of the image area, but the image area is usually sparse since it includes contributions only from a few processors. In later phases of the algorithm, as we move up the compositing tree, the processors are responsible for a smaller and smaller portion of the image area, but the sparsity decreases since an increasing number of processors have contributed image data. At the top of the tree, all processors have complete information for a small rectangle of the image. The final image can be constructed by tiling these subimages onto the display.

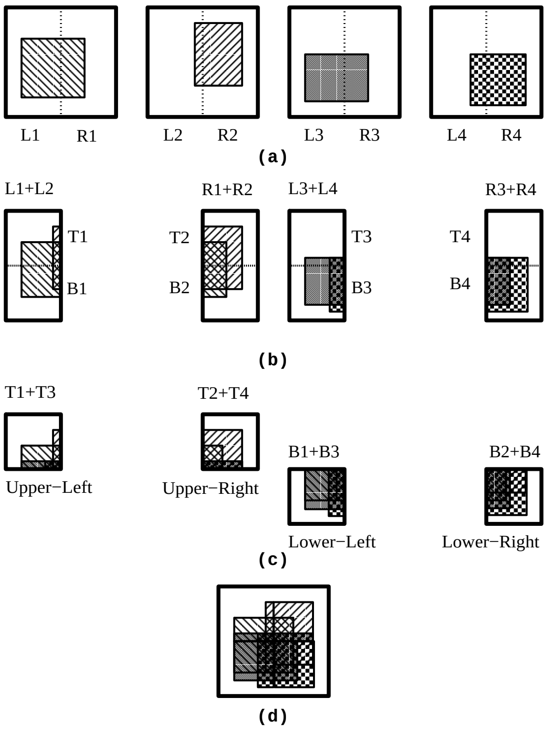

Figure 3: Parallel Compositing Process

The sparsity of the image can be exploited, since compositing need only occur where non-blank image data is present. Each processor maintains a screen aligned bounding rectangle of the non-blank subimage area. The processors only store and composite within this bounding rectangle. Two forces affect the size of the bounding rectangle as we move up the compositing tree: the bounding rectangle grows due to the contributions from other processors, but shrinks due to the subdivision of the image plane as we move up the tree. The net effect is analyzed in greater detail in the next section.

Figure 3 illustrates the *binary-swap* compositing algorithm graphically for four processors. When all four processors finish rendering locally, each processor holds a partial image, as depicted in (a). Each partial image is subdivided into two half-images by splitting along the X axis. In our example, as shown in (b), Processor 1 keeps only the left half-image and sends its right half-image to its immediate-right sibling, which is Processor 2. Conversely, Processor 2 keeps its right half-image, and sends its left half-image to Processor 1. Both processors then composite the half image they keep with the half image they receive. A similar exchange and compositing of partial images is done between Processor 3 and 4.

After the first stage, each processor only holds a partial image that is half the size of the original one. In the next stage, Processor 1 alternates the image subdivision direction. This time it keeps the upper half-image and sends the lower half-image to its second-immediate-right sibling, which is Processor 3, as shown in (c). Conversely, Processor 3 trades its upper half-image for Processor 1's lower half-image for compositing. Concurrently, a similar exchange and compositing between Processor 2 and 4 are done. After this stage, each processor hold only one-fourth of the original image. For this example, we are done and each processor sends its image to the display device. The final composited image is shown in (d). It has been brought to our attention that a similar merging algorithm has been developed independently by Mackerras [10].

In our current implementation, the number of processors must be a perfect power of two. This simplifies the calculations needed to identify the compositing partner at each stage of the compositing tree and ensures that all processors are active at every compositing phase. The algorithm can be generalized to relax this restriction if the compositing tree is kept as a *complete* (but not necessarily full) binary tree, with some additional complexity in the compositing partner computation and with some processors remaining idle during the first compositing phase.

3.4 Communications Costs

At the end of local rendering each processor holds a subimage of size approximately $p\ n^{-2/3}$ pixels, where p is the number of pixels in the final image and n is the number of processors [3]. The total number of pixels, over all n processors is therefore: $p\ n^{1/3}$ pixels. In the first phase of the binary-swap algorithm, half of the these pixels are communicated. At this point, each processor performs a single binary compositing step with the data from the neighboring processors, operating only on the nonempty pixels in the portion of the image plane assigned to it. Some reduction in the total number of pixels will occur due to the depth overlap that is resolved in this compositing stage.

On average, the reduction due to depth overlap is by a factor of $2^{-1/3}$ at each compositing phase. To see this, consider what happens over three compositing phases. The k-D tree partitioning of the data set will split each of the coordinate planes in half over 3 levels in the tree. Orthogonal projection onto any plane will have an average depth overlap of 2. For example, assume the viewpoint is selected so that we are looking straight down the Z axis. The X and Y splits of the data will result in *no* depth overlap in the image plane while the Z split will result in complete overlap, cutting the total number of pixels in half. Thus, over three compositing phases, the image size is reduced by a factor of 1/2. The average over each phase then is: $2^{-1/3}$, so that when three stages are invoked (cubing the per-stage factor) we get the required factor of 1/2.

This process repeats through $\log n$ phases. If we number the phases from $i=1$ to $\log n$, each phase begins with $2^{-(i-1)/3}\ n^{1/3}\ p$ pixels and ends with $2^{-i/3} n^{1/3}\ p$ pixels. The last phase therefore ends with $2^{-(\log n)/3}\ n^{1/3}\ p = n^{-1/3}\ n^{1/3}\ p = p$ pixels, as expected. At each phase, half of the pixels are communicated. Summing up the pixels communicated over all phases:

$$\text{pixels transmitted} = \sum_{i=1}^{\log n} \left(\frac{1}{2}\ 2^{-(i-1)/3}\ n^{1/3}\ p \right)$$

The $2^{-(i-1)/3}$ term accounts for depth overlap resolution. The $n^{1/3}\ p$ term accounts for the initial local rendered image size, summed over all processors. The factor of 1/2 accounts for the fact that only half of the active pixels are communicated in each phase. This sum can be bounded by pulling out the terms which don't depend on i and noticing that the remaining sum is a geometric series which converges:

$$\begin{aligned} \text{pixels transmitted} &= \sum_{i=1}^{\log n} \left(\frac{1}{2}\ 2^{-(i-1)/3}\ n^{1/3}\ p \right) \\ &= \frac{1}{2}\ n^{1/3}\ p \sum_{i=0}^{\log n-1} 2^{-i/3} \\ &\leq 2.43\ n^{1/3}\ p \end{aligned}$$

3.5 Comparisons with Other Algorithms

Other alternatives for parallel compositing have been developed simultaneously and independently of our work. One, which we will call *direct send*, subdivides the image plane and assigns each processor a subset of the total image pixels. This is the approach used by Hsu and Neumann [1, 3]. Each rendered pixel is sent directly to the processor assigned that portion of the image plane. Processors accumulate these subimage pixels in an array and composite them in the proper order after all rendering is completed. The total number of pixels transmitted with this method is $n^{1/3}\ p\ (1 - 1/n)$, as reported by Neumann [3]. Asymptotically this is comparable to our result, but with a smaller constant factor.

In spite of the somewhat higher count of pixels transmitted, there are some advantages of our method over direct send. Direct send requires that each rendering processor send its rendering results to, potentially, every other processor. Indeed, Neumann recommends interleaving the image regions assigned to different processors to ensure good load balance and network utilization. Thus, direct send compositing may require a total of $n(n-1)$ messages to be transmitted. In binary-swap compositing, each processor sends exactly $\log n$ messages, albeit larger ones, so the total number of messages transmitted is $n \log n$. When per-message overhead is high, it can be advantageous to reduce the total message count. Furthermore, binary-swap compositing can exploit faster *nearest neighbor* communications paths when they exist. Early phases of the algorithm exchange messages with nearest neighbors, and this is exactly when the number of pixels transmitted is largest, since little depth resolution has occured. On the other hand, binary-swap compositing requires $\log n$ communications phases, while the direct send method sends each partial ray segment result only once. In an asynchronous message passing environment, direct-send latency costs are $O(1)$, while in a synchronous environment they are $O(n)$, since the processor must block until each message is received and acknowledged. Binary-swap latency costs grow by $O(\log n)$, whether synchronous or asynchronous communications are used.

Camahort and Chakravarty have developed a different parallel compositing algorithm [2] which we will call the *projection* method. Their rendering method uses a 3D grid decomposition of the volume data. Parallel compositing is accomplished by propagating ray segment results front-to-back along the path of the ray through the volume to the processors holding the neighboring parts of the volume. Each processor composites the incoming data with its own local subimage data before passing the results onto its neighbors in the grid. The final image is projected on to a subset of the processor nodes; those assigned outer back faces in the 3D grid decomposition.

Like the other methods, the projection method requires a total of $O(n^{1/3}\ p)$ pixels to be transmitted. Camahort and Chakravarty observe that each processor sends its results to, at most, three neighboring processors in the 3D grid. Thus, by buffering pixels, the projection method can be implemented with only 3 message sends per processor as compared to $\log n$ for binary swap and $n-1$ for direct send. However, each final image pixel must be routed through $n^{1/3}$ processor nodes, on average, on its way to a face of the volume. This means that the messages latency costs grow by $O(n^{1/3})$.

4 Implementation of the Renderer

We have implemented two versions of our distributed volume rendering algorithm: one on the CM-5 and another on groups of networked workstations. Our implementation is composed of three major pieces of code: a data distributor, a renderer, and an image compositor. Currently, the data distributor is a part of the host program which reads data piece by piece from disk and distributes it to each participating machine. Alternatively, each node program could read its piece from disk directly if parallel I/O facilities exist.

Our renderer is a basic renderer and is not highly tuned for best performance. Data dependent volume rendering acceleration techniques tend to be less effective in parallel volume renderers, compared to uniprocessor implementations, since they may accelerate the progress on some processors more than others. For example, a processor tracing through empty space will probably finish before another processor working on a dense section of the data. We are currently exploring data distribution heuristics that can take the complexity of the subvolumes into account when distributing the data to ensure equal load on all processors.

For shading the volume, surface normals are approximated as local gradients using central differencing. We trade memory for time by precomputing and storing the three components of the gradient at each voxel. For example, a data set of size 256×256×256 requires more than 200 megabytes to store both the data and the precomputed gradients. This memory requirement prevents us from sequentially rendering this data set on most of our workstations.

4.1 CM-5 and CMMD 3.0

The CM-5 is a massively parallel supercomputer which supports both the SIMD and MIMD programming models [11]. The CM-5 in the Advanced Computing Laboratory at Los Alamos National Laboratory has 1024 nodes, each of which is a Sparc microprocessor with 32 MB of local RAM and four 64-bit wide vector units. With four vector units up to 128 operations can be performed by a single instruction. This yields a theoretical speed of 128 GFlops for a 1024-node CM-5. The nodes can be divided into partitions whose size must be a power of two. Each user program operates within a single partition. Our CM-5 implementation of the parallel volume renderer takes advantages of the MIMD programming features of the CM-5. MIMD programs use CMMD, a message passing library for communications and synchronization, which supports either a hostless model or a host/node model.

We chose the host/node programming model of CMMD because we wanted the option of using X-windows to display directly from the CM-5. The host program determines which data-space partitioning to use, based on the number of nodes in the CM-5 partition, and sends this information to the nodes. The host then optionally reads in the volume to be rendered and broadcasts it to the nodes. Alternatively, the data can be read directly from the DataVault or Scalable Disk Array into the nodes' local memory. The host then broadcasts the opacity/colormap and the transformation information to the nodes. Finally, the host performs an I/O servicing loop which receives the rendered portions of the image from the nodes.

The node program begins by receiving its data-space partitioning information and then its portion of the data from the host. It then updates the transfer function and the transform matrices. Following this step, the nodes all execute their own copies of the renderer. They synchronize after the rendering and before entering the compositing phase. Once the compositing is finished, each node has a portion of the image that it then send back to the host for display.

4.2 Networked Workstations and PVM 3.1

Unlike a massively parallel supercomputer dedicating uniform and intensive computing power, a network computing environment provides nondedicated and scattered computing cycles. Thus, using a set of high performance workstations connected by an Ethernet, our goal is to set up a volume rendering facility for handling large data sets and batch animation jobs. That is, we hope that by using many workstations concurrently, the rendering time will decrease linearly and we will be able to render data sets that are too large to render on a single machine. Note that real-time rendering is generally not achievable in such an environment.

We use PVM (Parallel Virtual Machine) [12], a parallel program development environment, to implement the data communications in our algorithm. PVM allows us to implement our algorithm portably for use on a variety of workstation platforms. To run a program under PVM, the user first executes a daemon process on the local host machine, which in turn initiates daemon processes on all other remote machines used. Then the user's application program (the node program), which should reside on each machine used, can be invoked on each remote machine by a local host program via the daemon processes. Communication and synchronization between these user processes are controlled by the daemon processes, which guarantee reliable delivery.

A host/node model has also been used. As a result, the implementation is nearly identical to that on the CM-5. In fact, the only distinct difference between the workstation's and CM-5's implementation (source program) is the communication calls. Basically, for most of the basic communication functions, PVM 3.1 and CMMD 3.0 have one-to-one equivalence.

5 Tests

We used three different data sets for our tests. The *vorticity* data set is a 256×256×256 voxel CFD data set, computed on a CM-200, showing the onset of turbulence. The *head* data set is the now classic UNC Chapel Hill MR head at a size of 128×128×128. The *vessel* data set is a 256×256×128 voxel Magnetic Resonance Angiography (MRA) data set showing the vascular structure within the brain of a patient.

Figure 4 illustrates the compositing process described in Figure 3, using the images generated with the *head* data set using eight processors. In Figure 4, each row shows the images from one processor, while from left to right, the columns show the intermediate images before each composite phase. The right most column shows the final results, still divided

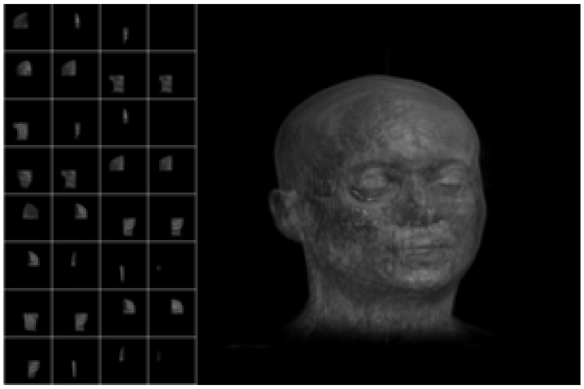

Figure 4: Head Data Set and Parallel Compositing Process

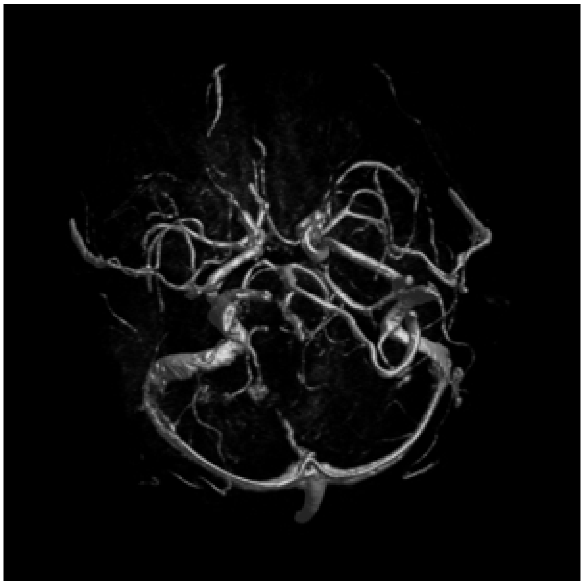

Figure 5: Vessel Data Set

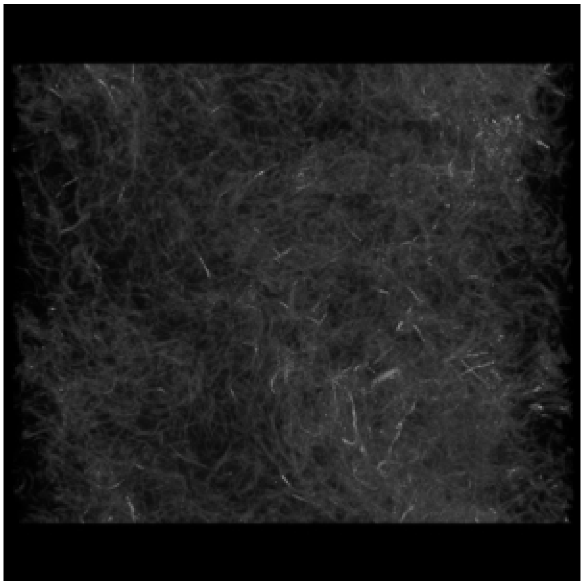

Figure 6: Vorticity Data Set

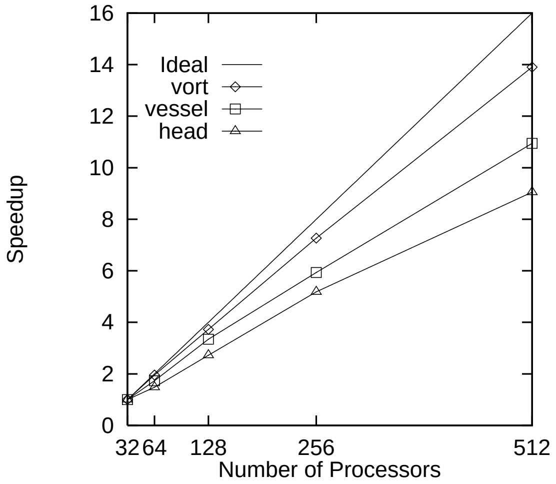

Figure 7: CM5 Speedup for 512 × 512 image size

function	32	64	128	256	512
dist	89.87	93.516	83.185	94.326	49.157
rend	48.2005	24.4303	12.697	6.3434	3.1878
comp	0.6309	0.5579	0.4091	0.3736	0.3213
comm	0.0843	0.0231	0.0181	0.0138	0.0097
send	0.9918	0.965	0.9645	1.0151	0.9849

Table 1: CM-5 Time (in seconds) Breakdown, Vorticity Data Set, 512 × 512 image size

among the eight processors. The final tiled image is blown up and displayed on the right. Figures 5 and 6 show images of the other two data sets rendered in parallel using the algorithm described here.

5.1 CM-5

We performed multiple experiments on the CM-5 using partition sizes of 32, 64, 128, 256 and 512. When these tests were run, a 1024 partition was not available. Figure 7 shows the speedup results for a 512x512 image on each data set. Note that the speedup is relative to the 32 node running time.

As there is no communication in the rendering step, one might expect linear speedup when utilizing more processors. As can be seen from the three speedup graphs, this is not always the case due to the load balance problems. The *vorticity* data set is relatively dense (i.e. it contains few empty voxels) and therefore exhibits nearly linear speedup. On the other hand, both the *head* and the *vessel* data sets contain many empty voxels which unbalance the load and therefore do not exhibit the best speedup.

Timing results are shown in Figure 8 (all data sets using an image size of 512x512) and Figure 9 (vessel data set at several image sizes). All times are given in seconds. The times shown in the graphs are the maximum times for all the nodes for the two steps of the core algorithm: the rendering step and the compositing step. Data distribution and image gather times are not included in the graphs.

Table 1 shows a time breakdown by algorithm component: data distribution (dist), rendering (rend), compositing computation time (comp), compositing communications (comm)

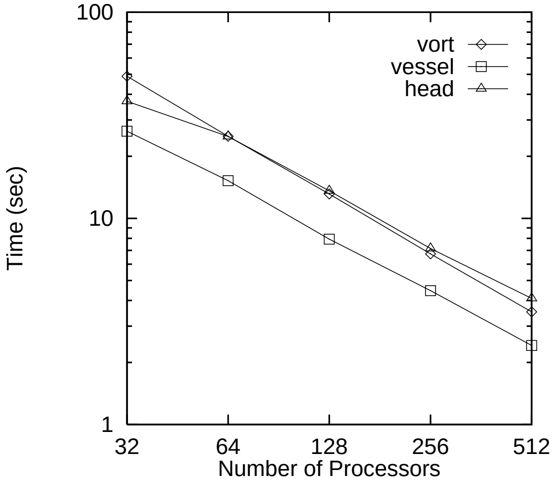

Figure 8: CM5 Run Times by Data Set, 512 × 512 image size

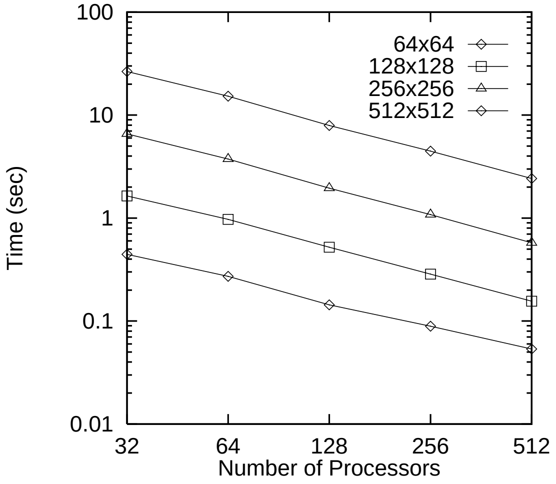

Figure 9: CM5 Run Times by Image Size, Vessel Data

and image gather (send) on a 512×512 rendering of the vorticity data. It is easy to see that rendering time dominates the process. It should be noted that this implementation does not take advantage of the CM-5 vector units. We expect much faster computation rates for both the renderer and compositor when the vectorized code is complete. The communication time varied from about 10 percent to about 3 percent of the total compositing time. As the image size increases, both the compositing time and the communication time also increase. For a fixed image size, increasing the partition size lowers the communication time because each node contains a proportionally smaller piece of the image and because the total communications bandwidth of the machine scales with the partition size.

The data distribution time includes the time it takes to read the data over NFS at Ethernet speeds on a loaded Ethernet. The image gathering time is the time it takes for the nodes to send their composited image tiles to the host. While other partitions were also running, the data distribution time could vary dramatically due to the disk and Ethernet contention. Taking the *vorticity* data set as an example, the data distribution varied from 40 to 90 seconds regardless of the partition size. Both of the data distribution time and image gathering time will be mitigated by use of the parallel storage and the use of the HIPPI frame buffer.

5.2 Networked Workstations

For our workstation tests, we used a set of 32 high performance workstations. The first four machines were IBM RS/6000-550 workstations equipped with 512 MB of memory. These workstations are rated at 81.8 SPECfp92. The next 12 machines were HP9000/730 workstations, some with 32 MB and others with 64 MB. These machines are rated at 86.7 SPECfp92. The remaining 16 machines were Sun Sparc-10/30 workstations equipped with 32 MB, which are rated at 45 SPECfp92. The tests on one, two and four workstations used only the IBM's because of their memory capacity. The tests with eight and 16 used a combination of the HP's and IBM's. The 16 Sun's were used for the 32 machine tests. It was not possible to assure absolute quiescence on each machine because they are in a shared environment with a large shared Ethernet and files systems. During the period of testing there was network traffic from network file system activity and across-the-net tape backups. In addition, the workstations lie on different subnets, increasing communications times when the subnet boundary must be crossed. Thus the communication performance was highly variable and difficult to characterize.

Timing results are shown in Figure 10 using all three data sets and an image size of 512x512. Again, data distribution and image gather times are not included in the graphs. In a heterogeneous environment, it is less meaningful to use speedup graphs to study the performance of our algorithm and implementation so speedup graphs are not provided.

For large images (e.g. 512×512) in the workstation environment, it is worthwhile to compress the subimages used in the compositing process. We have incorporated a compression algorithm into our communications library using an algorithm described in [13]. The compression ratio was about four to one, resuling in about 80% faster communication rates for the 32 workstation case. With fewer processors, computation tends to dominate over communications and compression is not as much of an advantage. The timing results show in Figure 10 include the effects of data compression.

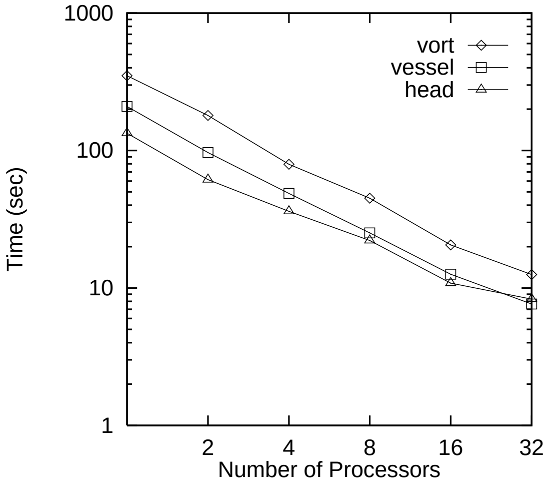

Figure 10: PVM Run Times by Data Set, 512 × 512 image size

function	1	2	4	8	16	32
rend	350.24	180.15	79.54	45.01	20.59	12.50
comp	0.03	0.17	0.09	0.10	0.12	0.12
comm	0.00	0.57	0.39	2.04	3.11	1.37

Table 2: PVM Time Breakdown (in seconds), Vorticity Data Set, 512 × 512 image size

Table 2 shows a time breakdown by algorithm component: rendering (rend), compositing computation time (comp), and compositing communications (comm). From the test results, we see that the rendering time still dominates when using eight or fewer workstations. It is also less beneficial to render smaller images due to the overhead costs associated with the rendering and compositing steps. Unlike the CM-5 results, tests on workstations show that the communication component is the dominant factor in the compositing costs. On the average, communication takes about 97% of the overall compositing time. On the CM-5, a large partition improved the overall communications time partly because the network bandwidth scales with the partition size. This is *not* true for a local area network such as an ethernet which has a fixed bandwidth available regardless of the number of machines used. On a LAN, communication costs of the algorithm *rise* with increasing numbers of machines.

The data distribution and image gather times varied greatly, due to the variable load on the shared Ethernet. The data distribution times varied from 17 seconds to 150 seconds while the image gather times varied from an average of .06 seconds for a 64×64 image to a high of 8 seconds for a 512×512 image. The above test results were based on Version 3.1 of PVM. Our initial tests using PVM 2.4.2 show a much higher communication cost, more than 70% higher.

In a *shared* computing environment, the communication costs are highly variable due to the use of the local Ethernet shared with hundreds of other machines. There are many factors that we have no control over that are influential to our algorithm. For example, an overloaded network and other users' processes competing with our rendering process for CPU and memory usage could greatly degrade the performance of our algorithm. Improved performance could be achieved by carefully distributing the load to each computer according to data content, and the computer's performance as well as its average usage by other users. Moreover, communications costs are expected to drop with higher speed interconnection networks (e.g. FDDI) and on clusters isolated from the larger local area network.

6 Conclusions

We have presented a parallel volume ray-tracing algorithm for a massively parallel computer or a set of interconnected workstations. The algorithm divides both the computation and memory load across all processors and can therefore be used to render data sets that are too large to fit into memory on a single uniprocessor. A parallel (*binary-swap*) compositing method was developed to combine the independently rendered results from each processor. The *binary-swap* compositing method has merits which make it particularly suitable for massively parallel processing. First, while the parallel compositing proceeds, the decreasing image size for sending and compositing makes the overall compositing process very efficient. Next, this method always keeps all processors busy doing useful work. Finally, it is simple to implement with the use of the k-D tree structure described earlier.

The algorithm has been implemented on both the CM-5 and a network of scientific workstations. The CM-5 implementation showed good speedup characteristics out to the largest available partition size of 512 nodes. Only a small fraction of the total rendering time was spent in communications, indicating the success of the parallel compositing method. Several directions appear ripe for further work. The host data distribution, image gather, and display times are bottlenecks on the current CM-5 implementation. These bottlenecks can be alleviated by exploiting the parallel I/O capabilities of the CM-5. Rendering and compositing times on the CM-5 can also be reduced significantly by taking advantage of the vector units available at each processing node. We are hopeful that real time rendering rates will be achievable at medium to high resolution with these improvements. Performance of the distributed workstation implementation could be further improved by better load balancing. In a heterogeneous environment with shared workstations, linear speedup is difficult. Data distribution heuristics which account for varying workstation computation power and workload are being investigated.

Acknowledgments

The MRA vessel data set was provided by the MIRL at the University of Utah. The vorticity data set was provided by Shi-Yi Chen of T-Div at Los Alamos National Laboratory. David Rich, of the ACL, and Burl Hall, of Thinking Machines, helped tremendously with the CM-5 timings. Alpha_1 and CSS at the University of Utah provided the workstations for our performance tests. Steve Parker helped in the analysis of binary-swap compositing by pointing out the factor of $2^{-(i-1)/3}$ due to depth overlap resolution. Thanks go to Elena Driskill and the reviewers for comments on earlier versions of this paper. This work has been supported in part by NSF/ACERC, NASA/ICASE and NSF (CCR-9210587). All opinions, findings, conclusions, or recommendations expressed in this document are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] W. M. Hsu, "Segmented Ray Casting for Data Parallel Volume Rendering," in *Proceedings 1993 Parallel Rendering Symposium*, pp. 7–14, 1993.

[2] E. Camahort and I. Chakravarty, "Integrating Volume Data Analysis and Rendering on Distributed Memory Architectures," in *Proceedings 1993 Parallel Rendering Symposium*, pp. 89–96, 1993.

[3] U. Neumann, "Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers," in *Proceedings 1993 Parallel Rendering Symposium*, pp. 97–104, 1993.

[4] B. Corrie and P. Mackerras, "Parallel Volume Rendering and Data Coherence," in *Proceedings 1993 Parallel Rendering Symposium*, pp. 23–26, 1993.

[5] K.-L. Ma and J. S. Painter, "Parallel Volume Visualization on Workstations," *Computers and Graphics*, vol. 17, no. 1, 1993.

[6] M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, pp. 29–37, May 1988.

[7] T. Porter and T. Duff, "Compositing Digital Images," *Computer Graphics (Proceedings of SIGGRAPH 1984)*, vol. 18, pp. 253–259, July 1984.

[8] J. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications ACM*, vol. 18, pp. 509–517, September 1975.

[9] H. Fuchs, G. Abram, and E. D. Grant, "Near Real-Time Shade Display of Rigid Objects," in *Proceedings of SIGGRAPH 1983*, pp. 65–72, 1983.

[10] P. Mackerras, "A Fast Parallel Marching Cubes Implementation on the Fujitsu AP1000," Tech. Rep. TR-CS-92-10, Department of Computer Science, Australian National University, 1992.

[11] Thinking Machines Corporation, *The Connection Machine CM-5 Technical Summary*, 1991.

[12] G. Geist and V. Sunderam, "Network-based Concurrent Computing on the PVM System," *Concurrency: Practice and Experience*, vol. 4, pp. 293–312, June 1992.

[13] R. Williams, "An Extremely Fast Ziv-Lempel Data Compression Algorithm," in *Proceedings of IEEE Computer Society Data Compression Conference* (J. Storer and J. Reif, eds.), April 1991.